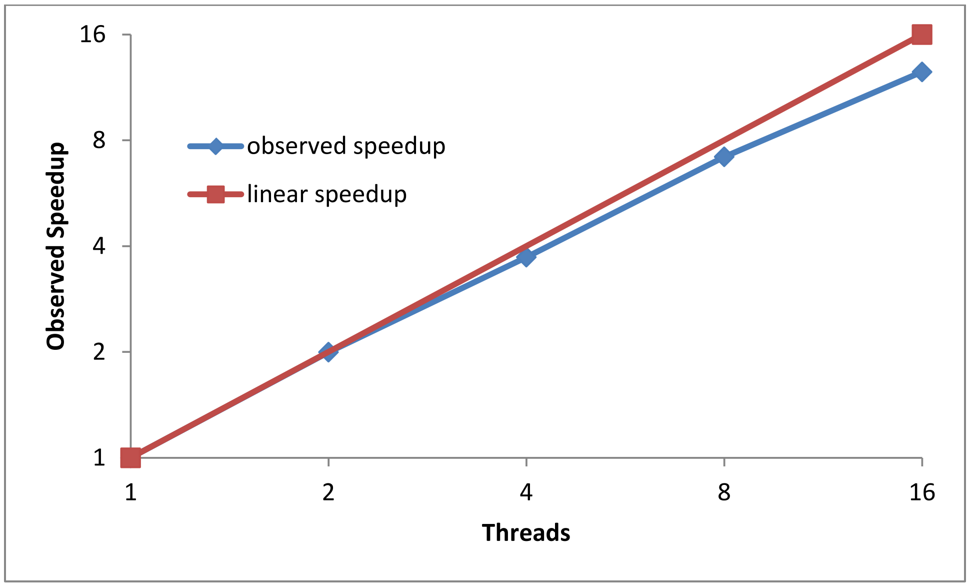
Ian Wilkes

Parallel Programming Assignment 2

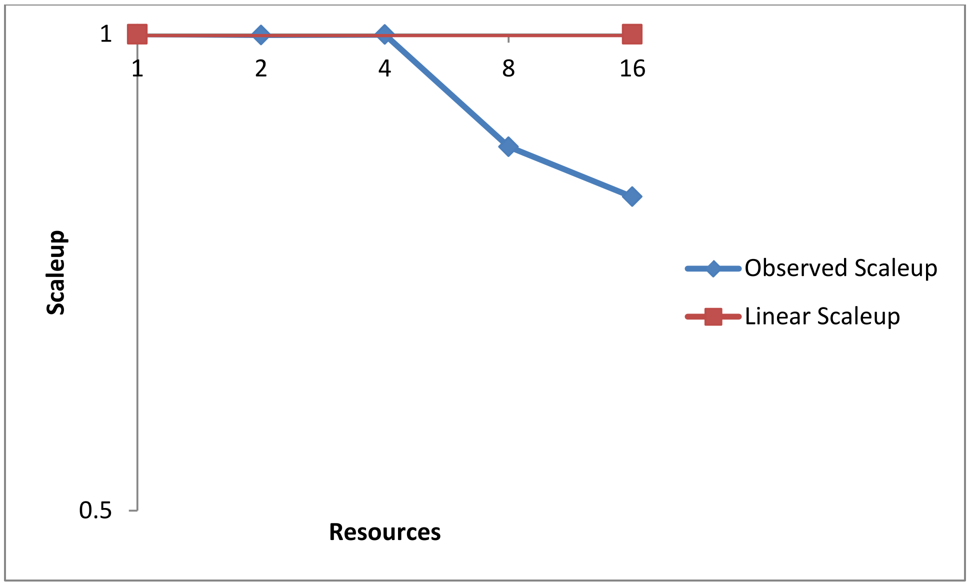
Ianm.wilkes@gmail.com

Part 1: Parallel Coin Flipping

1.a. Below are speedup and scaleup plots for the coin flipping experiment.



Above is a graph of the speedup of the coin flipping experiment. To produce this graph, the coin flipping code was run 20 times for numbers of threads from 1 to 16 on 10^9 flips.



Above is a graph of the scaleup of the coin flipping. For this experiment values of N from 1 to 16 were explored, on 10^8 * N flips respectively.

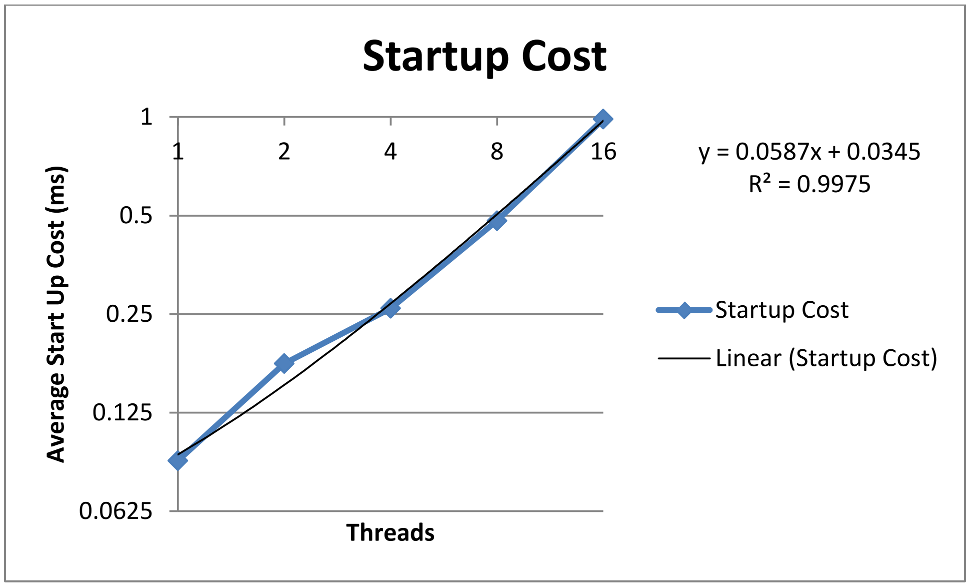
1.b. In this experiment the algorithmic speedup and scaleup should both be quite linear, as there is little to no skew, and the algorithm is only serialized on the final write of the number of heads checked found. Algorithmicly, there should be no cost in adding an additional thread, so the only startup cost or serial cost is that of iterating through the various for loops to start the threads, wait until they are done, and finally to sum the total of the number of heads found by each thread. For cases where the number of flips is large compared to the number of threads being used, here up to 8, this time is insignificant. Next, there should be no interference, as each thread has its own coin flipper, which is independent from the others, and thus only requires local data. Finally, skew can be seen to be almost zero, as an equal number of computations are required for all threads, as all threads are computing the same number of flips. Thus, essentially linear speedup and scaleup are achieved in the algorithmic sense.

The observed speedup and scaleup are both nonlinear, probably due to startup costs incurred in creating additional threads. As in the algorithmic sense, the skew should be zero, as each thread does the same amount of work on equivalent hardware, and the interference is minimal as there are no shared variables.

1.c. Speedup increases more slowly after additional cores past the 8 physical cores are used because the additional 8 cores are virtual, and created by hyperthreading. Hyperthreading does not cause the same performance benefits that adding an additional physical core would, so at this point the speedup increases more slowly. Hyperthreading is more efficient than running only one thread on a core at a time, so this is why when switching up past 8 cores speedup increases at a rate slower than the linear speedup which is observed on cores 1 through 8. The same is seen in the scaleup, as at higher numbers of cores, the scaleup drops from nearly 1 to a little bit over 75%, which indicates that the additional threads being run do not run as quickly as standard threads. Startup costs may also be influencing this data because we see a drop in scaleup at 8 cores, which is not seen at 1, 2, or 4 cores.

2. a. For my experiment, I loop over the section of the standard parallel coin flipping code, and measure the time required to allocate and close a certain number of threads 5000 times for 0 coin flips each time. For this, the startup costs are the costs associated with allocating and closing the treads, so that is what I have measured. To ensure that only the startup costs are measured, no coins are flipped in each stage, so all the time is taken is startup costs, as there can be no skew since all threads have nothing to do, and no interference as no variables are shared. The startup of threads is executed 5000 times to get a duration which is easily measurable, enabling us to

compare the differences between thread counts more accurately. From my experiment I have produced the following graph.

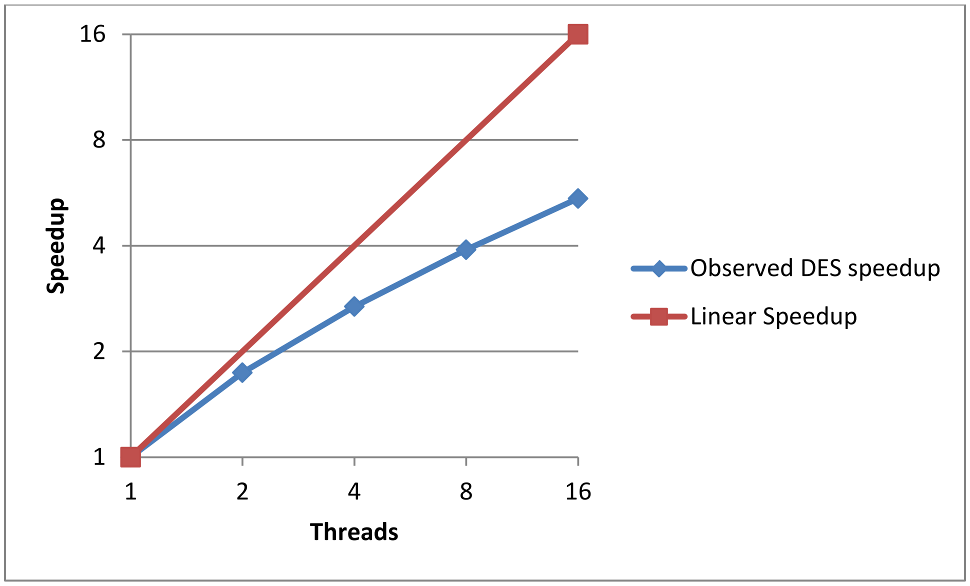


2. b. Startup cost seems to be approximately 0.06 ms per thread launched. This is gathered from a trend line of the average of 5000 runs launching and joining the specified number of threads. Each case was examined with twenty trials. It makes sense that startup costs would be a linear function of the number of threads being run, as additional threads cause additional bookkeeping and coordination costs for the OS to keep straight. This however does not provide the percentage of the time of a program which devoted to this cost, as that number would depend on both the number of threads being run, as well as the size of the parallel portion of the code. To get a percentage estimate, I compared my startup data with that of the speedup test, assuming that the startup cost was the same for both tests, and the parallel portion of the code was all of the other time in the speedup test, to determine the percentage of time which is devoted to startup costs as a function of the number of threads. Here, the problem is again 10^9 flips. Using the linear trend of those cases, we get the following formula: $Startup\ Cost\ Percentage\ (\%) = 0.000065n - 0.00013$ where n is the number of threads.

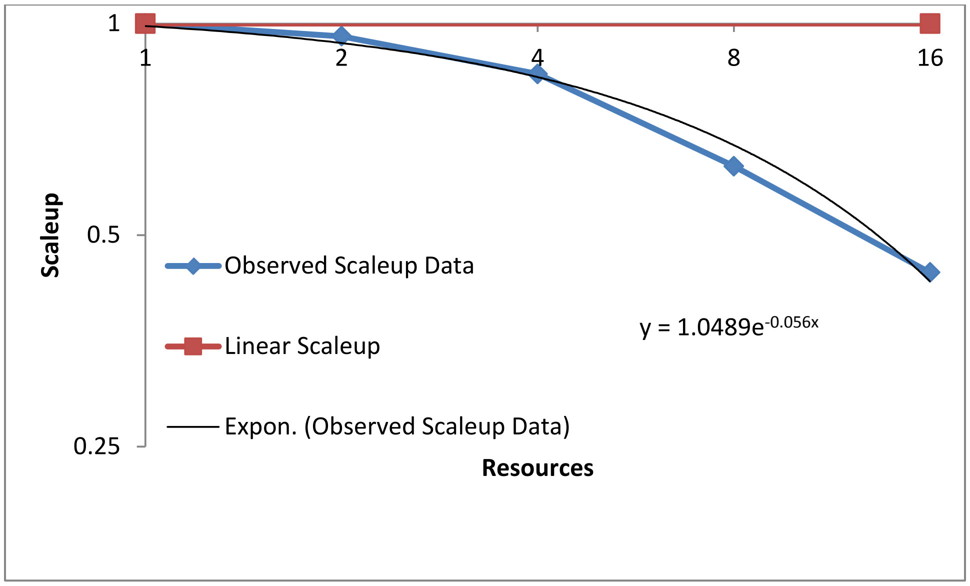
2. c. Using this formula to get 1 - P, we can predict the speedup of 100, 500 and 1000 threads using Amdahl's law, which are 61, 29 and 15 respectively for 10^9flips. The performance drops off, because the amount of time devoted to startup drives through the roof, while the computation time goes essentially to zero for each thread, thus leaving the serial portion to dominate.

Brute Force a DES Key

1.a. The following is the speedup plot for the DES Key breaker, using the average of 20 trials at each number of threads, and a 22 bit key in all cases.



The following is the corresponding scaleup chart, starting with a key of length 19 plus the log of the number of threads used.



Both the speedup and scaleup are sublinear.

1.b. Both the speedup and scaleup are sublinear, and it may be due to speedup and possibly skew, but not interference. My current implementation has no shared variables, so it does not seem possible to have interference. There may be some skew, as the searching for the index of a string within a string may not be constant time for every string encountered. The indexOf method may return early in some cases, reducing the computation of a given thread. More likely however, is that the drop in speeup and scaleup are due to growing startup costs for larger numbers of threads, as for each thread new encryption and decryption objects have to be created, in addition to encrypting the original message, and quite a few values need to be copied from one location to another. The longer the constructor takes, the more startup costs will be incurred. Because the curvature seems to be somewhat constant across the whole range of threads, it does not seem as if hyperthreading is the cause of the issue.

1.c. If we extrapolate the scaleup from the above curve using an exponential fit, which seems to match the data, and look at 64 cores, we get a scaleup value of 0.029. Now, to get the time required to compute a 25 bit key, or a value of 6 for n, we can use the definition of speedup and the amount of time required to calculate the 19 bit key on one machine, to get the time of 84.6 seconds. Now if we assume that without adding any more resources doubling the problem size doubles the time required to solve the problem, then we get the total time to compute a 56 bit DES key to be $84.6 * 2^{56-25} s = 1.82 * 10^{11}\ s$, or in other words approximately 5750 years.